The GISS GCM Good Programming Guide

Version 1.0 February 1999

Gavin Schmidt, Jeff Jonas, Jean Lerner & Ruedy Reto

The GISS GCM (and all attendant offshoots) has developed into a large distributed effort. This guide is an effort to help integrate GISS-wide good coding practices that improve the efficiency of the code, make it more transparent and hopefully (in the long term) lead to some degree of homogenization.

This guide will be split into a number of sections. Firstly, we will highlight some of the common examples of 'bad' code and indicate some better alternatives. These should become common sense programming habits. The next section deals with the ways of getting rid of unnecessary `GO TO` statements, and discusses more structured approaches. We then outline some of the more useful elements of `FORTRAN 90` which can be used to enhance the readability of the code. The first appendix describes tools that are under-utilised but can be very helpful. The other appendix contains some of the issues that arise in porting the model code to the SGI machines. Examples are given where we feel the correct behavior is not obvious.

This is not intended to be a comprehensive guide to FORTRAN (or to the GISS GCM). It is intended only to highlight some of the more common problems that occur in the model and should be seen as a useful reference for the programmers and scientists in the building.

Please let us know about any features that could be added to this document. They will appear in the next version (contact gavin@giss.nasa.gov).

What to definitely avoid

- Do not calculate invariant expressions at every time step. Put those calculations whose answer never changes in an `IFIRST` section and calculate them once only at the start of every run. Note that on the SGI machines, code segments containing an `IFIRST` section must be compiled with the `-static` option in order to save the local variables. However, unnecessary compilation with `-static` can add a lot of overhead, slowing down the code. For instance, `QUSBxxx` routines do not require any saving of local variables and hence need not be compiled with this option. As a better alternative, all needed local variables should be declared in a SAVE statement and the `-static` option never used. Care must be taken to make sure that all such variables have been identified before switching over.

- Avoid calculating expressions that do not depend on the loop variable inside the loop. Move such independent code outside the loop.

- Do not divide! Division takes many times longer than multiplication and so if you are dividing by the same denominator more than once, calculate the reciprocal and multiply by that. Sometimes the compiler can do this for you, but this cannot be relied upon.

- Organise your loops and arrays so that the innermost loop is for the first index. For instance, in an `I,J,L` loop over `T(I,J,L)`. `L` should be the outermost loop, followed by `J`, followed by `I`. Doing it any other way is highly inefficient. If your subroutine processes things by `(I,J)` grid box, and performs calculations in the vertical (such as `MSTCNV`, `CONDSE` etc), write the internal arrays with `L` as the first index.

  The most egregious example of bad looping occurs right at the start of subroutine `DYNAM`,

```
      COMMON U,V,T,P,Q
      COMMON/WORK6/UT(IM,JM,LM),VT(IM,JM,LM),TT(IM,JM,LM),PT(IM,JM),
     *             QT(IM,JM,LM)
      COMMON/WORK2/UX(IM,JM,LM),VX(IM,JM,LM),TX(IM,JM,LM),PX(IM,JM)
      ....
      DO 310 J=1,JM
      DO 305 L=1,LM*3+1
      DO 305 I=1,IM
      UX(I,J,L)=U(I,J,L)
  305 UT(I,J,L)=U(I,J,L)
      DO 310 L=1,LM
      DO 310 I=1,IM
  310 QT(I,J,L)=Q(I,J,L)
```

  On the face of it, there's nothing wrong. It works fine. However, if rewritten properly the model actually runs noticeably faster. What happened here was that the code probably started out correct just processing UX and UT. Then someone must have added QT and decided that they could save some coding by rearranging. (This is theory–the oldest code that could be found was from 1985, and that has the error. Unfortunately, `MB112M9.S` can't be fixed because there are too many `.U` files to check for conflicts.) Anyhow, a couple of years ago Jean went through all the dynamics routines for the 31-layer model and rewrote all the inside-out loops. It ran 15-18% faster when finished.

  As a general rule, common blocks should not be initialised like this since it is not transparent. See the example in the `FORTRAN 90` section for the preferred method.

- Avoid exponentiation (ie. `A**5`). All polynomials should be calculated using Horner's Rule (i.e. `A*X**3 + B*X**2 + C*X + D` should be coded as `D + X*(C + X*(B + X*A))`. Factorize as much as possible.

- Calls to mathematical functions (`log, sin, cos,` etc.) should be minimised. These are expensive. If repeated calculations are needed, calculate them once and then save the result.

- Use constants for DO loops (not variables). Prior to the PARAMETER statement (in FORTRAN 77), IM, JM, LM, and related quantities were in common blocks. Now, IM, JM, and LM are defined as parameters, and replaced by placeholders (IM0, JM0, LM0) in the common block. When loops use constants, the loops are set up by the compiler, instead of on the fly at execution time. Quantities in common blocks are treated as variables, not constants, by the compiler so there is overhead setting up the loop. So DO loops will run faster with IM and JM as loop variables, rather than IM0, JM0.

  However, there are still people using JMM1 and LMM1 (variables). Instead, JM-1 and LM-1 should be used (which could not be done pre-FORTRAN 77). These are constants and are treated as such by the compiler. As a general point, all constants should be declared in a parameter statement.

Further information on optimization is available in the "XL Fortran: Optimization Guide" handbook or the IRIX Fortran manual.

Unstructured habits

Much of the GCM code (and some of the programmers!) dates back to the punch-card days of yore. In contrast to the current fashion for 1960's nostalgia, we believe that the code really should reflect some of the advances made by FORTRAN 77 (and maybe even FORTRAN 90!). Many of the most confusing examples are driven by a desire to write compact code. While this was an issue with punch-cards, it is no-longer so important. However, if the desire for compact code is still present, we recommend you use the new features of FORTRAN 90 which allow extremely compact code to be written in clear and unambiguous ways (see next section).

- GOTO bad, IF (..) THEN good. Prior to the introduction of BLOCK IF structures (and now DO...WHILE loops) huge numbers of GOTO statements were needed. However, there is a tendency for GOTO structures to become very spaghetti-like, and it becomes very difficult to follow the logic of the code. Replacing GOTO with any of the other structures makes it much clearer for the reader (and for the compiler).

- Initialisation across a common block using only one array is bad habit. If the common block changes or is rearranged, this can cause serious confusion. See the example in the FORTRAN 90 section for a clearer way of doing this efficiently. (Also see last comment in this section)

- Obsolete code should be deleted. Use a new version number for new code (do not just call it 'new' or 'latest'). If you need a record, you can always look at the previous version stored on Ipcc2 or Ra. Commenting it out only leads to confusion. Leaving it in, and not using it, is inefficient and confusing to the next generation of programmers.

- Over-use of equivalence statements. While equivalence statements are useful for writing compact code, their use should really be restricted for operations that are applied uniformly over the equivalenced arrays (such as initiallisation, input/output etc.)

- `IF ... GOTO` branches out of a `DO` loop. While this is supported by our compilers and is technically correct, it is not a recommended feature of standard fortran. Problems have occured with compilers/optimizers that did not treat it correctly. In particular, if a loop variable is used outside the loop subsequent to such a branch, this sometimes gave incorrect results. This practice should be avoided if possible. More generally, these branches seriously inhibit proper optimization (since loop structures cannot be changed). Use the `DO..WHILE` construction instead (see next section).

- Do not use active code lines as continue statements. (This makes Jean CRAZY!) Constructs like this

  ```
      do 110 i=1,im
      b = something complicated
  110 a = something complicated depending on b
  ```

  are not actually wrong, but they are very awkward if someone wants to do some debugging or add an additional calculation. Then, instead of inserting a statement, they also have to rewrite:

  ```
      do 110 i=1,im
      b = something complicated
      a = something complicated depending on b
      c = something depending on a
  110 continue
  ```

  This is a pain in the neck!!! It also makes it harder to see what's going on when you do a diff on two files, because it looks like two things were changed, not one. Please use `CONTINUE` or `END DO` statements. (In fact with `DO ... END DO`, you can dispense with line numbers entirely).

- `COMMON` block cautions. Two rules of clean programming that are frequently violated in the GCM are i) out-of-bounds array referencing (as alluded to above), and ii) declaring a common block to be different sizes in different routines. While neither of these are strictly illegal, they do cause problems with optimisers/compilers. For instance, on the SGI the compiler option `-OPT=reorg_common` is on by default. This pads out common block to make them more efficient (by making sure arrays are on the same page of memory for instance). This is now turned off in `setup`. Other problems can also occur. For instance, when the compiler encounters common blocks of different sizes it will select the largest, except when the common block is initialised via a block data section even if there are larger references elsewhere. If we reduce our dependence on these kinds of violations, future problems are likely to be minimised.

Common blocks that are used in sub-modules (such as the PBL or tracer codes) are sometimes required in `MAIN, INPUT`, etc. It is much more straightforward to put these in separate (named) files which are then `INCLUDE`-d in the code. That way revisions and changes can be quickly accommodated.

New structures from `FORTRAN 90`

Many of the new features in `FORTRAN 90` significantly reduce the amount of coding needed and makes the code much more readable. There is much more to `FORTRAN 90` then we can do justice to here, but we particularly want to highlight the array processing facilities. To use `FORTRAN 90` constructs, you need only change `f77` to `f90` in your compilation scripts. There is one major caveat: the GCM II' as a whole does not run if compiled with `f90`. Some parts of it (the radiation?) cannot be compiled, but others have included `f90` features with no problems. Gary's model can be run with `f90`. Please ensure that any routines you modify will compile with `f90` prior to including any new constructs. Please report any problems (and solutions!) you uncover.

- Array arithmetic. In `FORTRAN 90`, arrays can be used directly in an expression (as long as it is conformal) without having to loop over the indexes. The example discussed above can be compactly written as below with the compiler deciding the most efficient way to loop over the variables.

```
      COMMON U,V,T,P,Q
      COMMON/WORK6/UT(IM,JM,LM),VT(IM,JM,LM),TT(IM,JM,LM),PT(IM,JM),
     *             QT(IM,JM,LM)
      COMMON/WORK2/UX(IM,JM,LM),VX(IM,JM,LM),TX(IM,JM,LM),PX(IM,JM)
      DIMENSION UALL(IM,JM,LM*3+1),UTALL(IM,JM,LM*3+1),UXALL(IM,JM,LM*3+1)
      EQUIVALENCE (UALL,U),(UTALL,UT),(UXALL,UX)
      ....
      UXALL=UALL
      UTALL=UALL
      QT=Q
```

  Another example is when you wish to only set a limited number of indexes, or ranges of the index. For this, : denotes the entire range of an index or `2:7` for instance denotes just the range 2 until 7 (inclusive). For instance, commonly the polar boxes are set to be the same and equal to the value at `I=1`. This could be written:

```
      P(2:IM,JM) = P(1,JM)
```

- New constructs: `CASE` and `DO...WHILE`. The `CASE` construct allows you to branch to any number of sub-sections. It is similar to an arithmetic `IF` or a computed `GOTO`, but is much more flexible. For instance,

```
SELECT CASE (ITYPE)
CASE (1)                        ! ITYPE = 1
   stuff
CASE (2:4)                      ! ITYPE = 2,3 or 4
   stuff
CASE DEFAULT                    ! ITYPE = anything else
   PRINT*,"ITYPE must be 1,2,3 or 4"
END SELECT
```

  The `DO...WHILE` construct can replace convoluted constructions involving `IF` and `GOTO` statements. In particular, it should definitely replace constructs that include `IF` statements branches out of `DO` loop. For example,

```
DO WHILE (Q(1) < QSAT)
  EVAP = ....
  Q(1) = Q(1) + EVAP
END DO
```

One last thing...

COMMENTS ARE GOOD! Please, please, please comment any code you write. It doesn't need to be very verbose, but important calculations, break points, branches etc. should be flagged. At some stage, other people will try to work through it - please think of them! In particular, messages printed out before a stop due to unrealistic conditions should be verbose rather than cryptic.

Appendix I: Helpful tools

The `dbx` debugger

`dbx` is a powerful debugging utility that can be used quickly to diagnose some common problems. If you have a core dump, then go to the run directory and type

```
dbx program.exe core
```

At the prompt, typing 'where' will give the routine and line number of where it died. Compiling with the `-g` option increases the amount of information available, i.e. 'print variable' will display the value of the variable at the time of the core dump.

The KAP preprocessor

A Preprocessor rewrites your fortran code before sending it to the compiler and can be very useful in identifying bottlenecks and inefficiencies in your code. Using the preprocessor can lead to significant improvements in time (average 10-15% faster for Gary's model). It is called via an option to the fortran compiler but is not available on all machines.

Appendix II: Modifications necessary for SGI computers

1. All `FORTRAN` and `C` utilities have to be recompiled.

2. Most ksh scripts are ok, some have to be modified - test them. Please be sure to add `#!/bin/ksh` at the top of all scripts, so that those who have different login shells can use them.

3. `xlf` .... has to be replaced by `f77` .... (or `f90`) There has to be a space between `-o and name-of-executable-file`

   OPTIONS:

   in model routine compilations the options `-n32 -static` or `-64 -mips4 -static` are needed (the 2nd gives better performance).

   `-static` guarantees that local variables in a subroutine are saved after returns, i.e. `IFIRST`-constructions should work.

   `-old_rl` should be used with direct-access files to make it compatible with the IBMs

   `-OPT:reorg_common=off` is needed with the new release

   `-mp` (in the link step) will allow large jobs to run much faster, if your .profile contains the line:
   `export PAGESIZE_DATA=64; export PAGESIZE_STACK=64; export PAGESIZE_TEXT=64`

   `-Wl,-woff,134 -Wl,-woff,15` arguements for `f77/f90` used when linking (in `setup` or `mkexe`) removes the pointless warnings, but leaves any important ones.

4. Optimizations (`-O3` cannot be used for individual subroutines for `-n32`) what worked so far is:
   `f77 -n32 -static -O2 -c xyz.f (--> xyz.o)` or
   `f77 -64 -static -O3 -c xyz.f (--> xyz.o)`
   for more speed:
   `f77 -64 -mips4 -static -O3 -OPT:fold_arith_limit=1409`
   `-O2` is almost as fast and safer than `-O3` (optimization level)
   link options:
   `-mips4 lfastm` (uses library of fast math.routines)

5. Differences in `FORTRAN` (missing/obsolete features etc):

   - `READ(...,NUM=nbytes)` is unavailable
   - `BLOCK DATA` have to be named
   - 'open' cannot be the name of a `COMMON BLOCK`
   - `T,F` cannot be used to initialize logical variables, unless `PARAMETER (T=.TRUE.,F=.FALSE.)` is added
   - Use `STOP` rather than `RETURN` in MAIN.
   - The function `ERFC` only takes real*4 args, use `DERFC` for real*8 args.
   - The compiler warns if you use `CALL SUBR(A,..)` where A is a variable whereas A is an array in the subroutine (using of course only A(1))
   - If the arrays and their dimensions are passed to a subroutine, the dimensions have to be passed also in each entry that passes the arrays (the original `HNTRP` does not work on the SGIs)
   - The following construction does NOT work (with or w/o -static)

     ```
     SUBROUTINE ABC (v1,...)
     (no v1=... statement)
     ENTRY XYZ (v1 is NOT an argument)
     expression involving v1          (result unpredictable if
                                       reached after CALL XYZ)
     ```

     Quick (but ugly) fix:

     ```
     SUBROUTINE ABC (v1xxx,...)
     v1=v1xxx
     ```

6. `NAMELIST READ('string',NML=...)` is unavailable ; namelists have to be read from external files: back to 66-version (using unit 8 instead of 14)

7. System calls to save rsf, acc files did not work reliably. Use simple `OPEN` instead as in model II' (`MB112M9.S`).

8. Model-specific changes:

   (a) remove all `READ(....NUM=...)` instances (`INPUT`,`readt` in `UTIL..`)

   (b) replace internal `NAMELIST` reads by reads from unit 8 (`INPUT`,2x)

   (c) name all `BLOCK DATA` (diagnostics, rad.routines, input,soils)

   (d) replace `RANDU`-subroutine (add `RANDSGI`)